

# **Network Virus Detection Using Associative SIMD Processors**

**Kenneth Atchinson  
Kevin Schaffer  
Dr. Robert A. Walker**

ASC Processor Group  
Computer Science Department  
Kent State University

# Outline

- **Introduction**
- Structure of a File Virus
- Virus Detection on Storage Media and Network
- Algorithms for Detecting Viruses
- Implementation Choices
- Implementation
- System Performance
- Conclusion
- Future Work

# Introduction


- Today's reality – computer viruses exist
- Over 70,000 viruses have been discovered, but only a few are in the wild, and a smaller number that cause significant damage
- Defending systems from computer viruses has become top priority in maintaining system functionality and data integrity
- To effectively *defend*, virus *detection* is the key
- Viruses live in files that we use – this is where we must look for them

# Outline

- Introduction
- **Structure of a File Virus**
- Virus Detection on Storage Media and Network
- Algorithms for Detecting Viruses
- Implementation Choices
- Implementation
- System Performance
- Conclusion
- Future Work

# Normal Program File

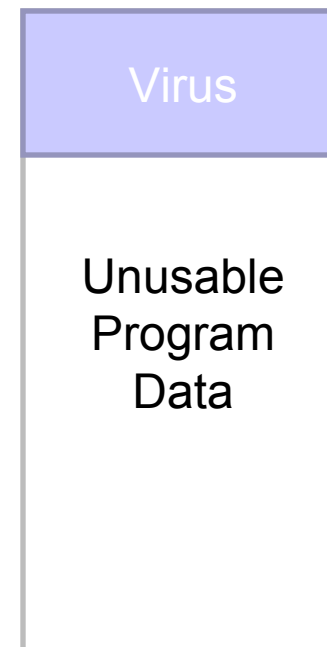
- **Typical uninfected program file contains executable instructions and data variables**
- **Viruses attach themselves to program files while they are in main storage (e.g. hard disk), infecting them**
- **Multiple ways viruses can attach to program files**



Uninfected  
Program

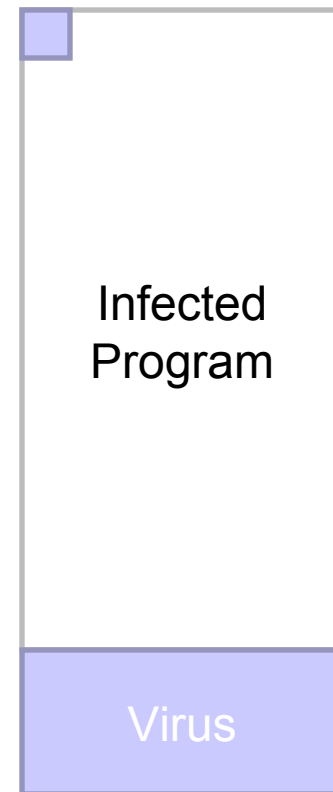
# Overwriting Virus

- **Virus code is written to the beginning of the program file**
- **Typically destroys the original program since the virus overwrites the original program code**
- **This virus doesn't change file size, thus harder to catch unless you "look" for the virus code**



# Appending Virus

- Virus code is appended to the end of the program file
- A small stub is written at the top of the program file, so that the virus is executed at the start of the program
- Stub spawns virus code off as another process, then continues with original program
- This virus increases the size of the program file, so visible detection is possible by simply looking for larger program files



# Trojan

- Entire program file is a virus, which is pretending to be a useful program
- File size more than likely is different than the original program file
- Original program code is destroyed and can never be recovered



Virus



# Observations

- A program file that has a virus is said to be an *infected file*
- Regardless of structure, a virus in a program file can be recognized by the code in which it contains
- Virus code can be expressed as a *pattern*, similar to regular expressions
- Detection is done via pattern recognition algorithms, where known virus code is compared with program file
- Detection program (Anti-Virus) is called a *Virus Scanner*

# Observations

- Sample virus code of known viruses is compressed and optimally stored for Virus Scanner in what is called *Virus Signature*
- Virus Signatures are stored in a *Virus Definition File* (a database of virus signatures)
- A look at where we currently detect viruses
  - Detection on Storage Media
  - Detection over a Network

# Outline

- Introduction
- Structure of a File Virus
- **Virus Detection on Storage Media and Network**
- Algorithms for Detecting Viruses
- Implementation Choices
- Implementation
- System Performance
- Conclusion
- Future Work

# Detection on Storage Media

- **Infected files are stored on the file system, virus detection occurs inside of disk blocks**
- **Virus detection on storage media is I/O intensive since we have to read disk blocks and scan them for virus patterns**
- **Virus detection is a single threaded process where program file contents are analyzed against all virus signatures**
- **Time to detect virus tends to be long, but not a factor since there is not any time constraints for completion**

# Detection over a Network

- **Infected files are transmitted over a network, thus virus detection occurs inside of the TCP/IP packet payload**
- **Virus detection over a network uses the same methods/algorithms employed by their storage media detection designs counterparts, however**
  - Time to detect virus is shorter since data must be transmitted quickly (minimizing store/forward delay)
  - Reduce number of patterns to check, thus detection is not as thorough as on storage media
- **I will explore this area in my research**

# Outline

- Introduction
- Structure of a File Virus
- Virus Detection on Storage Media and Network
- **Algorithms for Detecting Viruses**
- Implementation Choices
- Implementation
- System Performance
- Conclusion
- Future Work

# Virus Detection Algorithms

- Is *Exact Match* the only solution?
- Viruses can be polymorphic: the virus code is slightly modified during program file infection as to make it harder to detect
- Typical polymorphism for virus code is injection of NOOPs in code, increasing virus code size without changing virus code operation
- Virus signatures must account for the possibility of virus polymorphism
- Two detection algorithms: Sequence Alignment and Longest Common Subsequence (LCS)

# Sequence Alignment

- Procedure of comparing 2 or more sequences
- Searches series of individual character pattern in the same order in the sequence

```
GGHSRLILSQLGEEG.RLLAIDRDPQAI AVAKT
|||:::| : |::| ||:::||||:|:|:::
GGHAERFL.E.GLPGLRLIGLDRDPTALDVAR
```



# Sequence Alignment

- **Issue: If sequences do not “line up” exactly, detection will provide a false negative (e.g. Polymorphism)**

```
GGHSRLILSQLGEEG.RLLAIDRDPQAI(A)VAKT  
: : : : : : : : : : : : : : | : : : : : : : : : : : : : :  
KKGGHAERFL.E.GLPGLRLIGLDRDPTALDVA
```

- ACGTTACGTA~~AACTACTAGTACACACACTACCCAT~~  
ACAGTGTTAAGTAA~~ACTAACTAGTACACACGTACCCAT~~

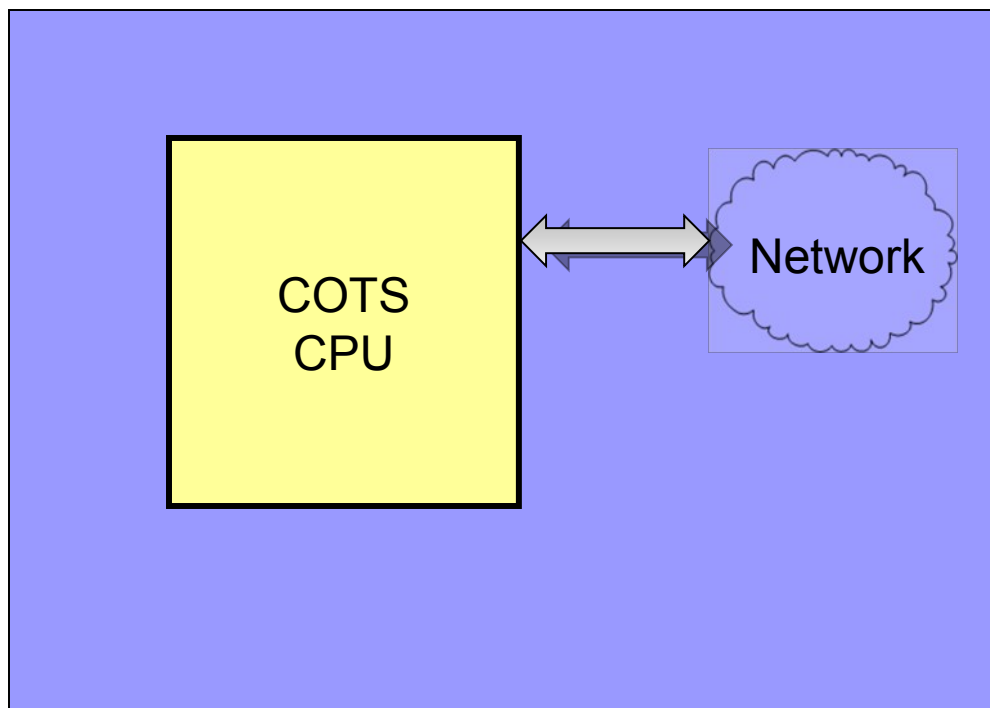
- Kenneth Atchinson - Kevin Schaffer - Robert A. Walker*

# Outline

- Introduction
- Structure of a File Virus
- Virus Detection on Storage Media and Network
- Algorithms for Detecting Viruses
- **Implementation Choices**
- Implementation
- System Performance
- Conclusion
- Future Work

# Implementation Choices

- **Implement a generic hardware design using Commercial Off the Shelf Components (COTS)**
  - Use a general purpose CPU
  - All work done via programs
  - Software emulating hardware = slow
- **Implement a custom hardware design**
  - Custom hardware provides speedup to execute algorithm in required time frame
  - Use Field Programmable Gate Arrays (FPGAs) instead of Application Specific Integrated Circuit (ASIC) to implement hardware
  - Hardware harder to change than software



## COTS CPU

**COTS CPU based design  
running virus detection  
algorithm**

**COTS CPU also handles  
network I/O**

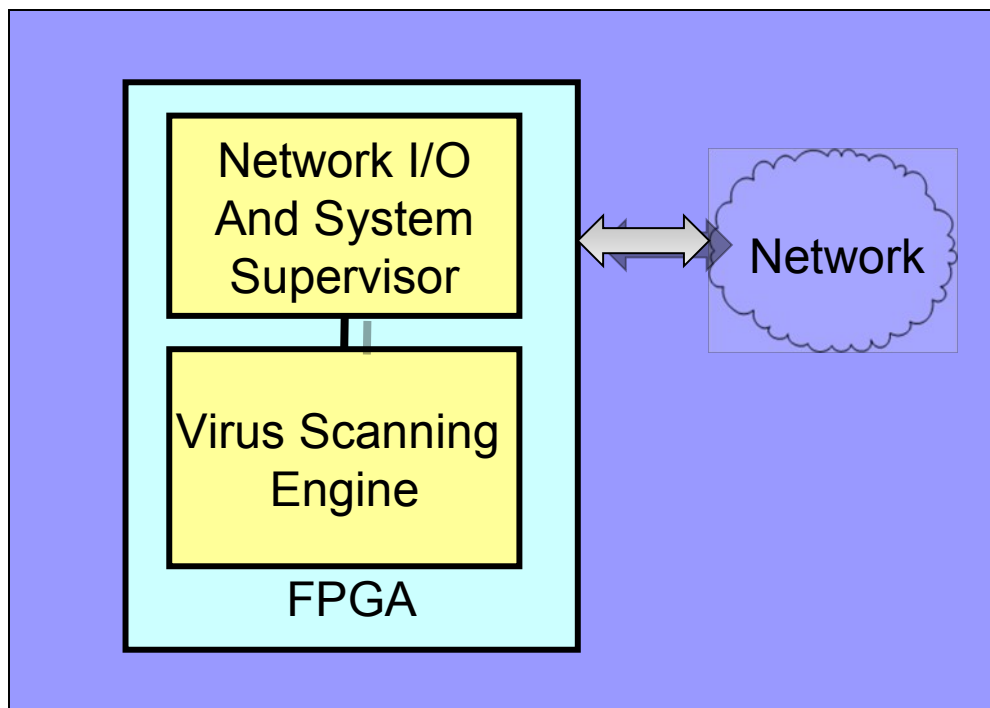
## Advantages

**Flexibility of system achieved via software**

## Disadvantages

**Costly solution – expensive COTS CPU**

**Slowest system due to COTS CPU being overloaded with virus  
scanning and handling network I/O**



## **Dedicated Hardware**

**Complete System on a Chip**

**Virus Scanner implemented in FPGA**

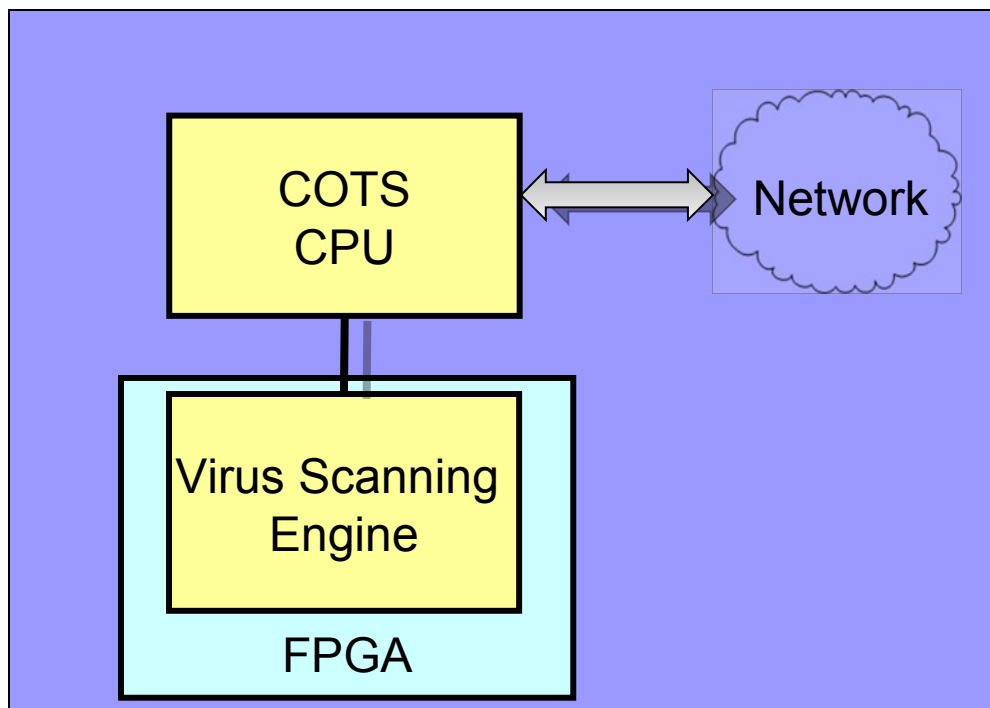
**FPGA also has to manage Network I/O**

## **Advantages**

**Very customized design, optimized for performance**

## **Disadvantages**

**Least flexible to runtime changes**



## COTS CPU with Coprocessor

**Semi-Custom Design of COTS Components and FPGA**

**COTS CPU to handle high order processing of data**

**Virus scanner implemented in FPGA, speeding up virus detection**

## Advantages

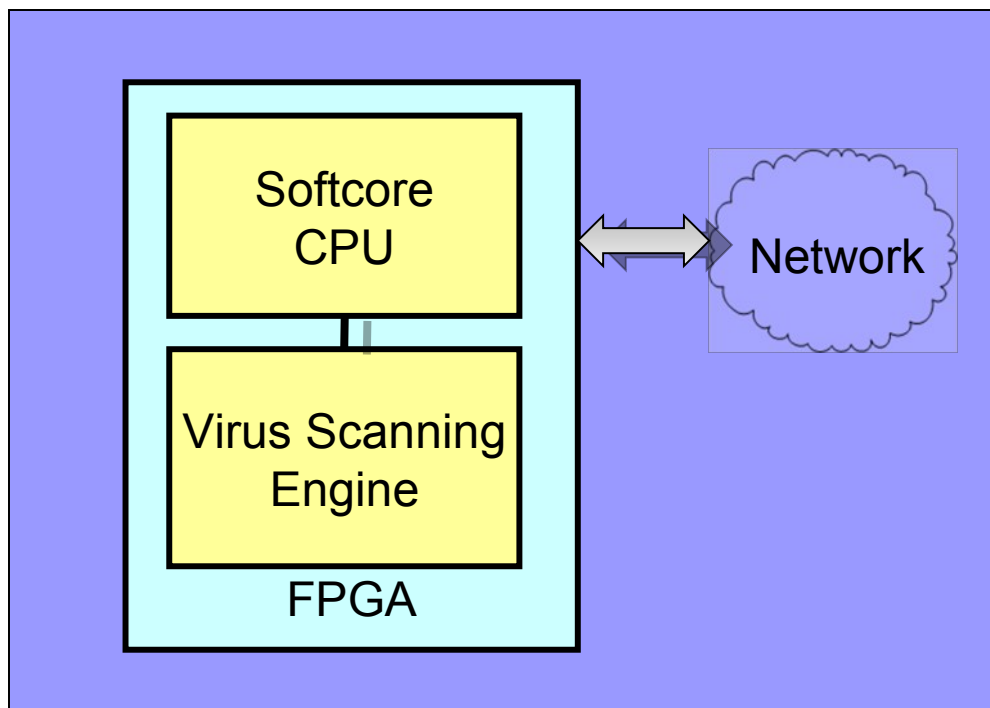
**COTS CPU gives design flexibility via software**

**Virus Scanning Engine speeds up virus detection**

## Disadvantages

**I/O Bottleneck – data to/from Virus Scanning Engine**

**Expensive – system is expensive CPU + FPGA**



## **Softcore CPU with Coprocessor**

**Complete System on a Chip**

**General Purpose CPU  
implemented in FPGA**

**Virus Scanner implemented  
in FPGA**

## **Advantages**

**Softcore CPU – design with just enough processor for the job**

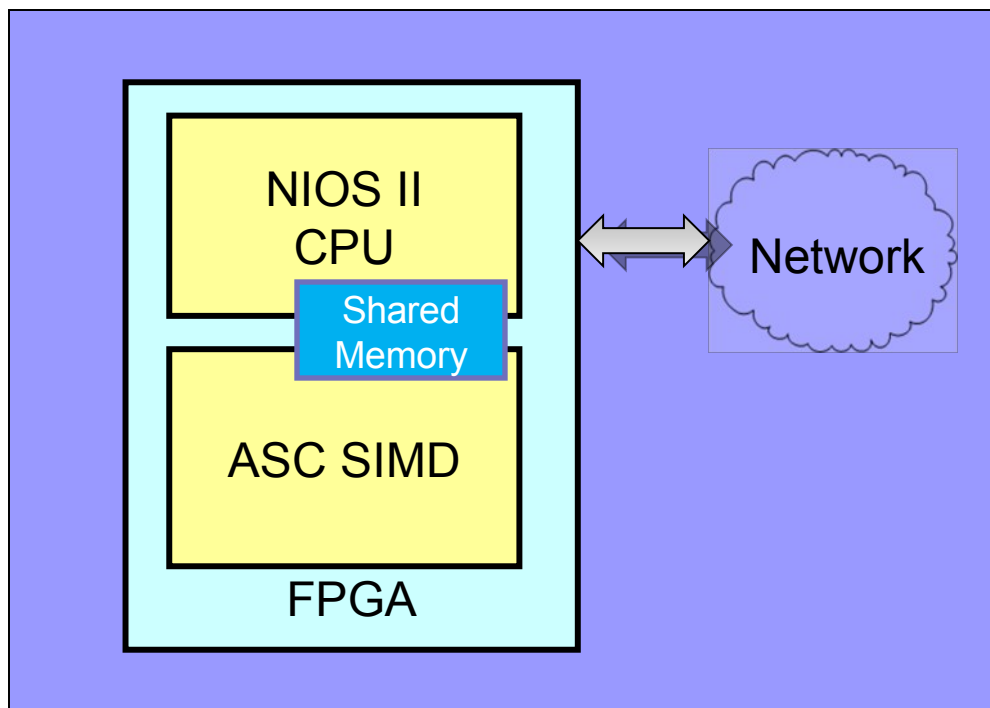
**Virus scanner engine can be specifically designed/programmed to  
detect particular viruses**

**Cost effective – all in one System on a Chip**



# Outline

- Introduction
- Structure of a File Virus
- Virus Detection on Storage Media and Network
- Algorithms for Detecting Viruses
- Implementation Choices
- **Implementation**
- System Performance
- Conclusion
- Future Work



## My Implementation

**Altera FPGA**

**NIOS II Softcore processor**

**Associative SIMD processor**

**Shared memory between  
NIOS II and Associative  
SIMD processors**

**Linux OS Environment**

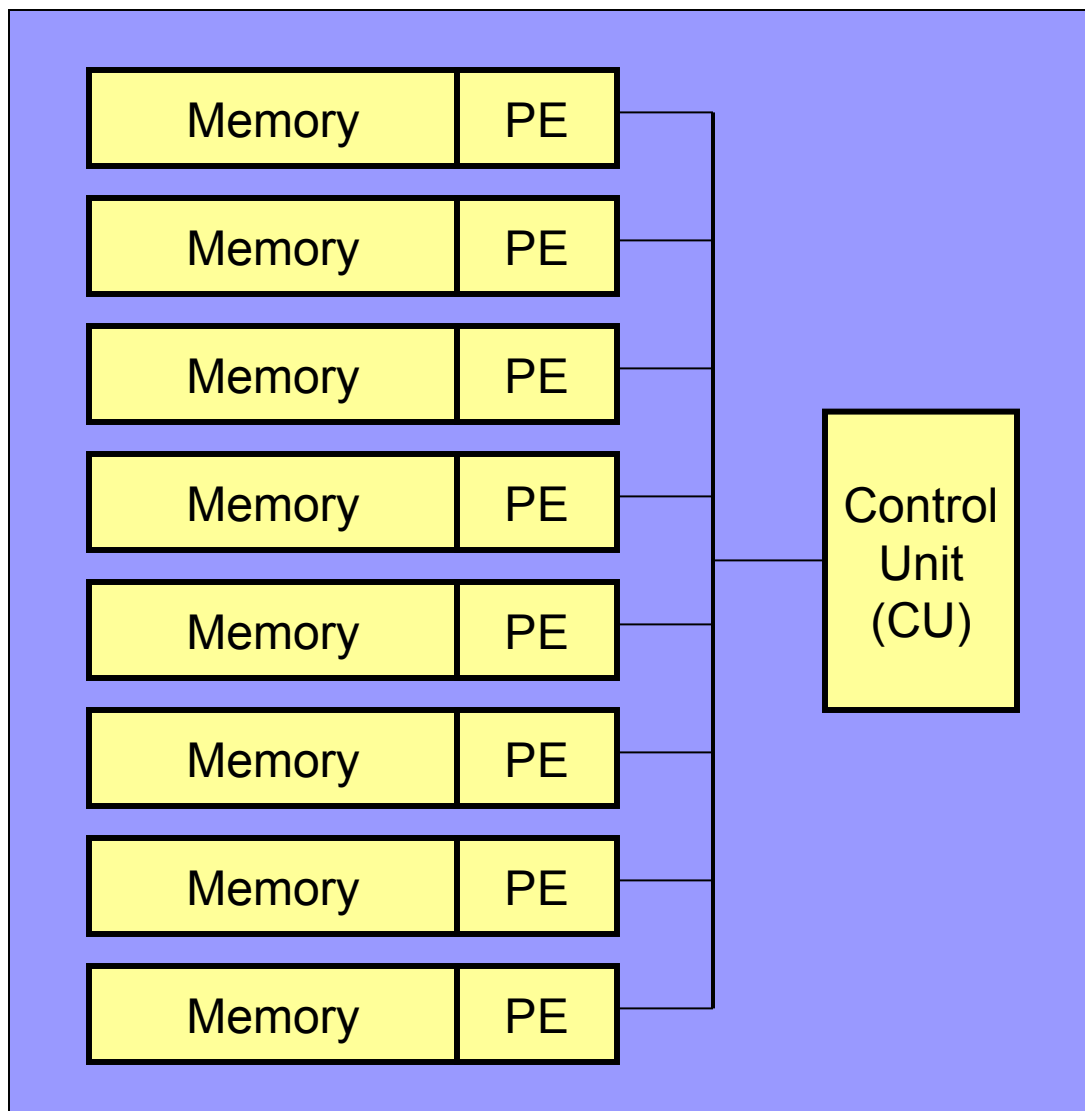
**C programming support**

## Advantages

**Kent State Associate SIMD (ASC) – modification of conventional SIMD architecture that uses specially designed Associative PEs**

**NIOS II + Linux + C – proven development/support environment**

**Shared memory tightly couple processors, reduces I/O bottlenecks**



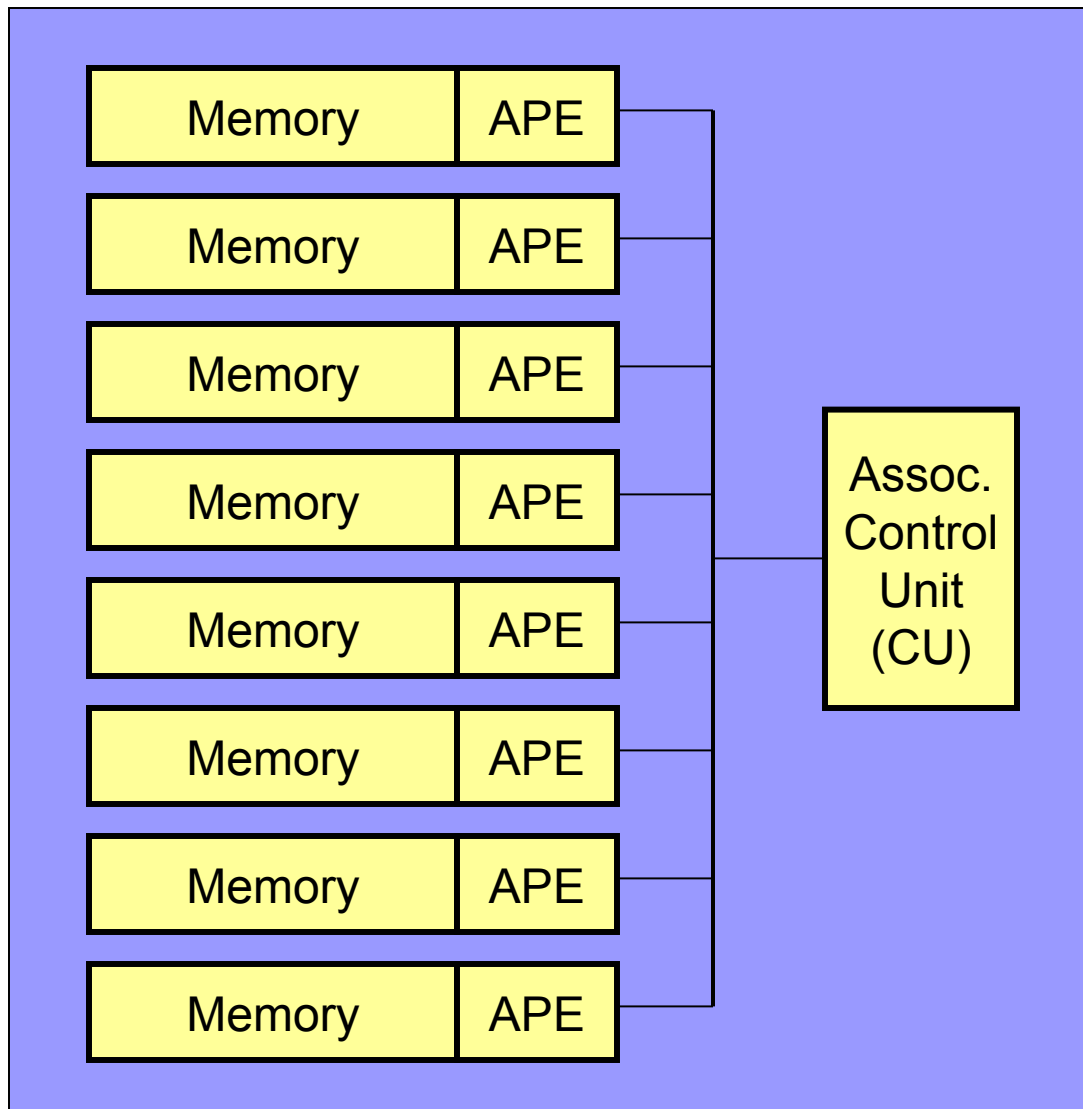
**Typical SIMD Array**

## Single Instruction Multiple Data (SIMD) Architecture

A Processing Element (PE) is a simple Processor (ALU, Registers) with private memory store

PE elements receive instructions from Control Unit

A single instruction is then executed over multiple (different) data



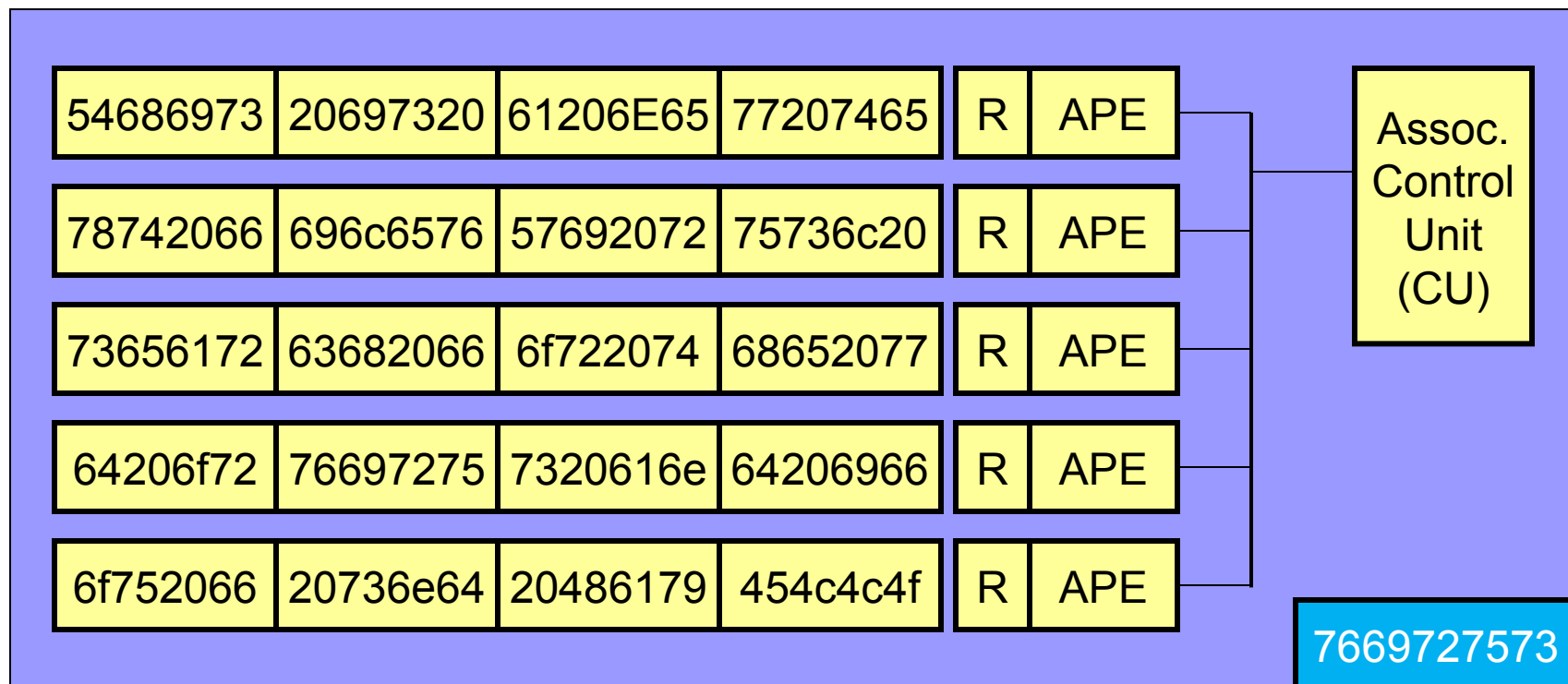
**Associative SIMD Array**

## Associative SIMD (ASC)

Each PE is an Associative PE, where PEs have a special responder bit

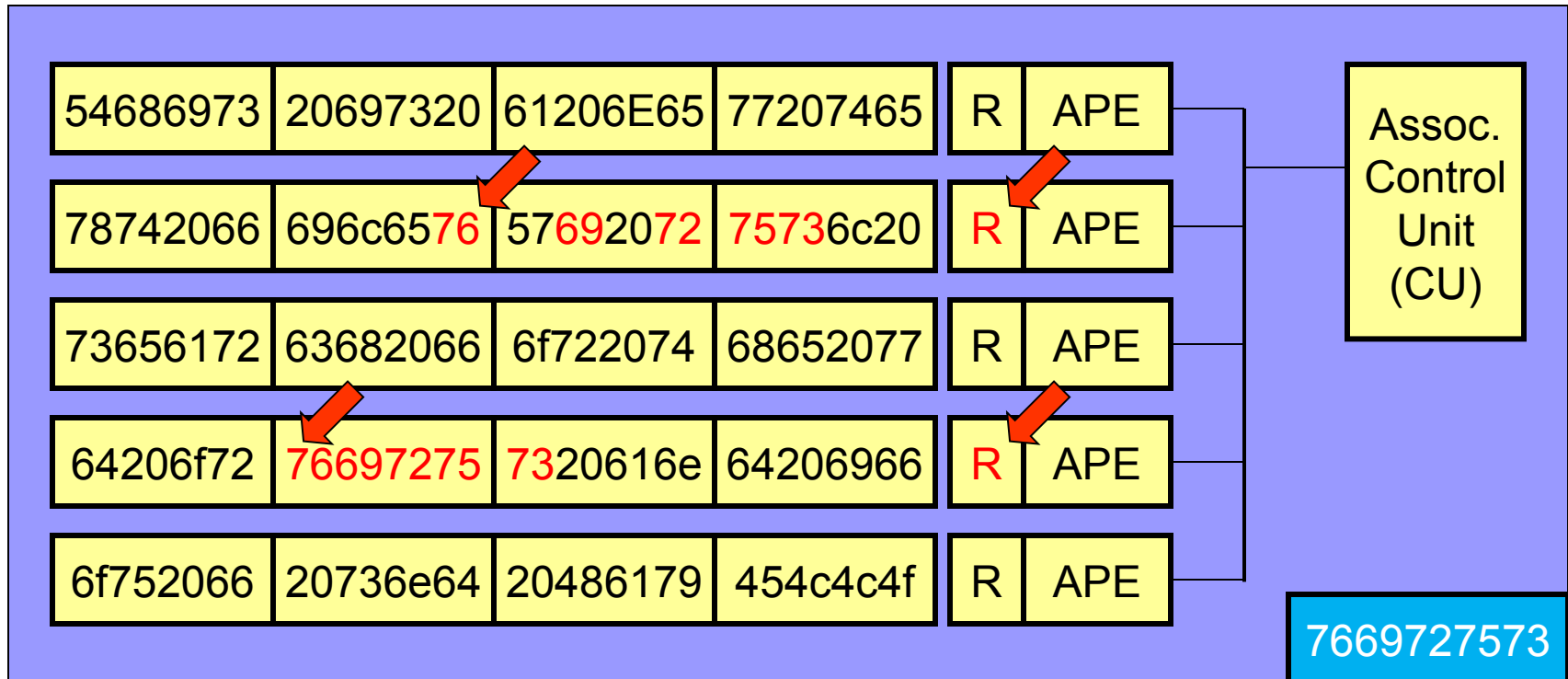
The responder bit can then be used to selectively engage/not engage PE to continue executing instructions from Control Unit

# Associative Search using LCS



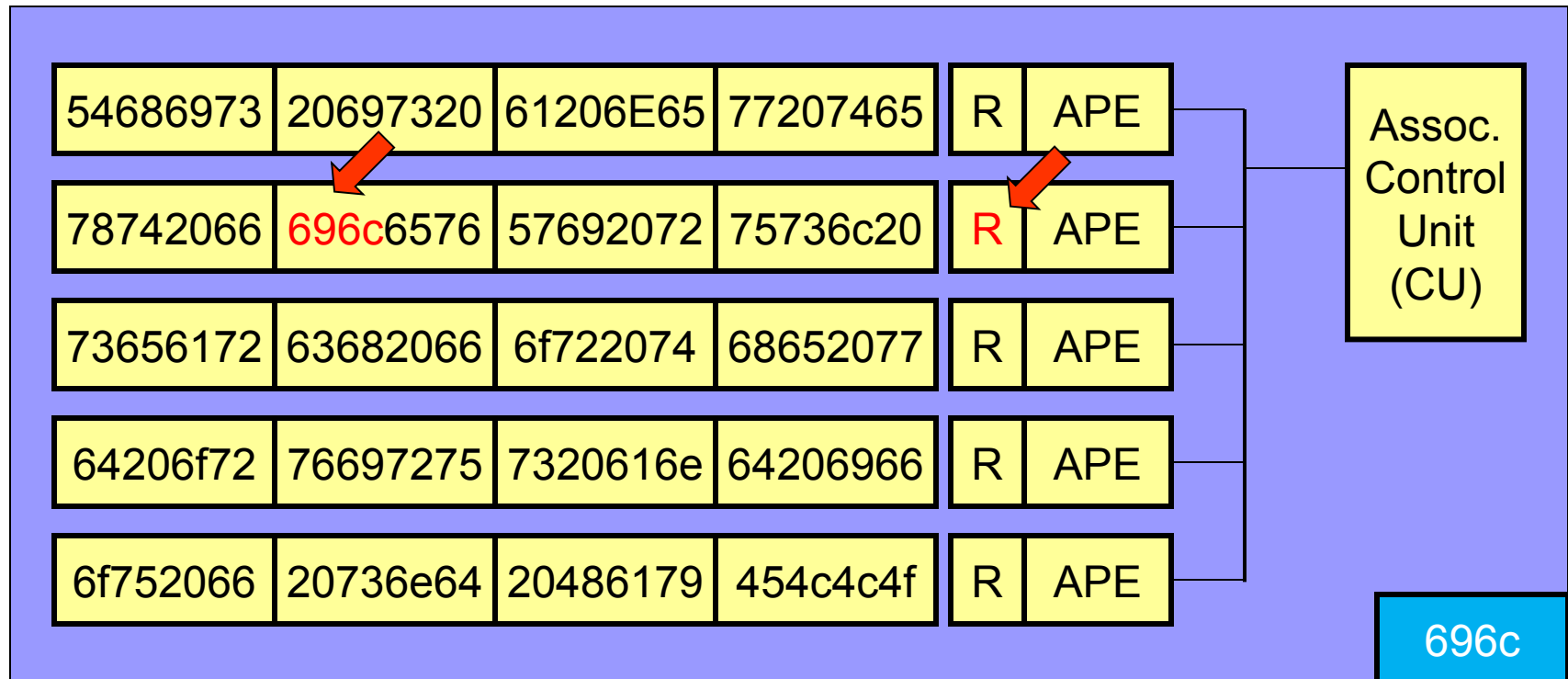
Each APE is loaded with a TCP/IP payload data, and performs a Longest Common Subsequence search. LCS algorithm set to search for data **7669727573**.

# Associative Search using LCS



**APEs that have a successful search will set the “R” responder field.**

# Associative Search using LCS



**Now for responders only, search for 696c. Only responders to second search will set “R” bit.**

# Outline

- Introduction
- Structure of a File Virus
- Virus Detection on Storage Media and Network
- Algorithms for Detecting Viruses
- Implementation Choices
- Implementation
- **System Performance**
- Conclusion
- Future Work



# Algorithm Performance

- LCS algorithm runs in  $O(nm)$  time, where  $n$  is the length of the text string, and  $m$  is the length of pattern string
- If we have  $K$  text strings to check, our run time is  $O(knm)$
- If we have  $K$  parallel processors, then our run time is back to  $O(nm)$ , a savings of  $O(k)$

# Outline

- Introduction
- Structure of a File Virus
- Virus Detection on Storage Media and Network
- Algorithms for Detecting Viruses
- Implementation Choices
- Implementation
- System Performance
- **Conclusion**
- Future Work

# Conclusion

- Viruses a growing threat to computer security
- Network virus detection increasingly becoming an important part of system security
- Using custom hardware solves timing requirements
- Using FPGAs gives flexibility in changing design to meet problem
- Using Associative SIMD processors solves pattern detection problem in parallel
- Speedup improvement of  $O(k)$  for  $K$  element SIMD

# Outline

- Introduction
- Structure of a File Virus
- Virus Detection on Storage Media and Network
- Algorithms for Detecting Viruses
- Implementation Choices
- Implementation
- System Performance
- Conclusion
- **Future Work**

# Future work

- **Implement KSU PE VHDL code with Altera NIOS II code**
  - Use NIOS II Softcore Processor to perform high order functions of getting data from network for analysis
  - NIOS II will run uCLinux, a Linux OS
  - NIOS II will be able to access KSU PE memory to enable fast data loading (remove I/O bottlenecks)
- **Implement Serial LCS on KSU PEs**
  - Determine “best” LCS match for this application. **Note:** “best” may not be longest (longer matches improves detection accuracy)

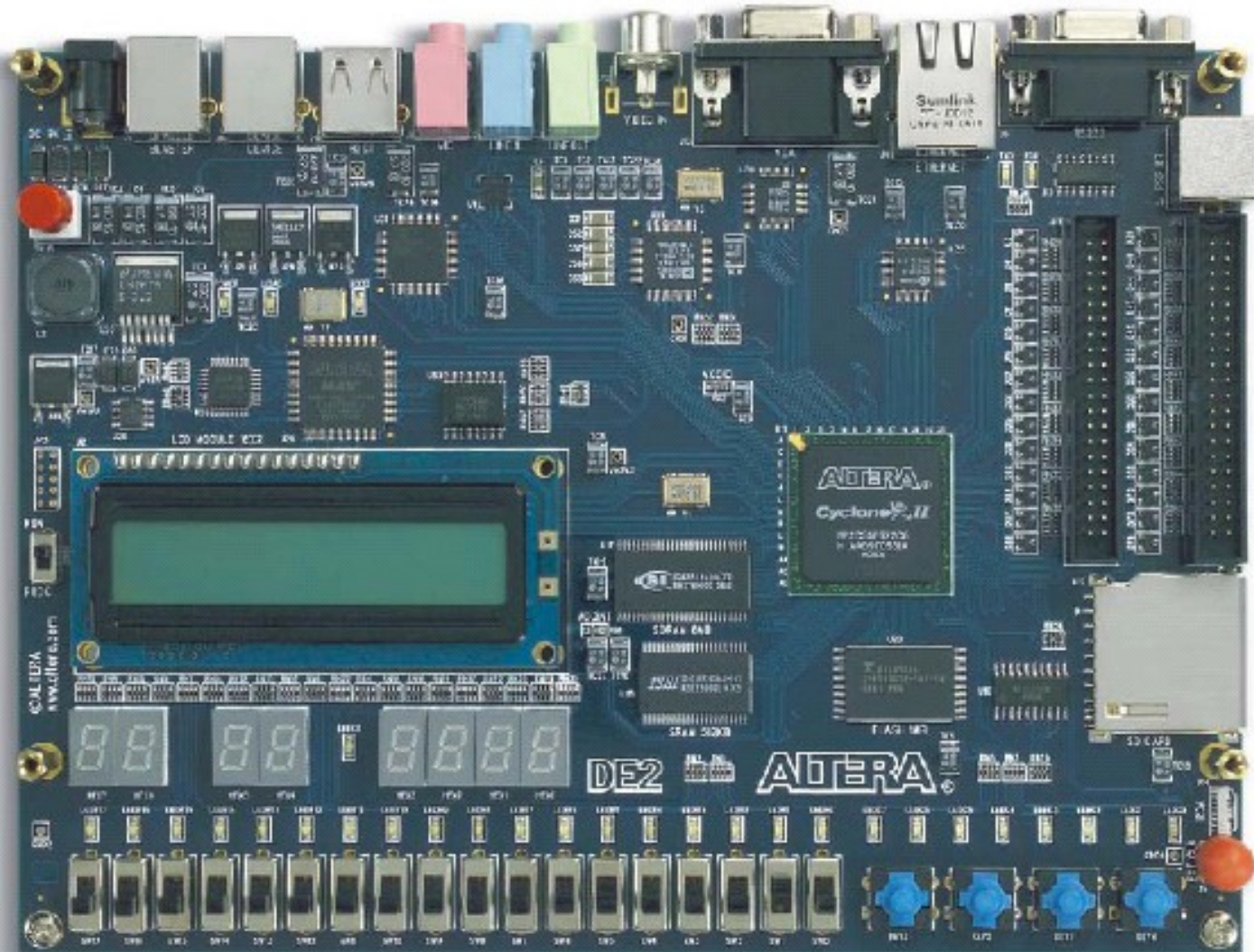
# Future work

- **Experiment with target quantity vs. pattern quantity searches**
  - Target quantity - tests lots of data against one pattern simultaneously
    - One virus signature constant in all PE memories, multiple TCP/IP payloads are written to PE memories
  - Pattern quantity - tests one data against lots of patterns simultaneously
    - Multiple virus signatures are constants in PE memories, write TCP/IP payload in all PE memories to check

# Questions



# Altera DE2 Development Board





# Quartus SOPC Builder

Altera SOPC Builder - nios\_0.sopc (C:\Documents and Settings\katchins\Desktop\work\DE2\_PE\nios\_0.sopc)

File Edit Module System View Tools Nios II Help

System Contents System Generation

Target: Device Family: Cyclone II

Clock Settings

Name	Source	MHz
clk	External	50.0

Add Remove

Use	Connec...	Module Name	Description	Clock	Base	End	IRQ
<input checked="" type="checkbox"/>		cpu_0	Nios II Processor	clk			
		instruction_master	Avalon Master	clk			
		data_master	Avalon Master	clk			
		jtag_debug_module	Avalon Slave	clk			
<input checked="" type="checkbox"/>		jtag_uart_0	JTAG UART	clk	0x01004000	0x01004007	1
		avalon_jtag_slave	Avalon Slave	clk			
<input checked="" type="checkbox"/>		sdram_0	SDRAM Controller	clk	0x00000000	0x007fffff	
		s1	Avalon Slave	clk			
<input checked="" type="checkbox"/>		tri_state_bridge_0	Avalon-MM Tristate Bridge	clk			
		avalon_slave	Avalon Slave	clk			
		tristate_master	Avalon Tristate Master	clk			
<input checked="" type="checkbox"/>		cfi_flash_0	Flash Memory (CFI)	clk	0x00800000	0x008fffff	
		s1	Avalon Tristate Slave	clk			
<input checked="" type="checkbox"/>		timer_0	Interval Timer	clk	0x00900000	0x0090001f	2
		s1	Avalon Slave	clk			
<input checked="" type="checkbox"/>		sysid	System ID Peripheral	clk	0x00900020	0x00900027	
		control_slave	Avalon Slave	clk			
<input checked="" type="checkbox"/>		uart_0	UART (RS-232 Serial Port)	clk	0x00900040	0x0090005f	3
		s1	Avalon Slave	clk			
<input checked="" type="checkbox"/>		timer_1	Interval Timer	clk	0x00900060	0x0090007f	4
		s1	Avalon Slave	clk			
<input checked="" type="checkbox"/>		lcd_16207_0	Character LCD	clk	0x00900030	0x0090003f	
		control_slave	Avalon Slave	clk			

Add... Remove Edit... Move Up Move Down Address Map... Filter...

